Architectural Optimization of the Monad Layer 1 Blockchain

Nikhil Ranjan
nikhil@polkassembly.io

The evolution of decentralized smart contract platforms has reached a critical bottleneck where the historical design choices of the Ethereum Virtual Machine (EVM) severely limit transaction execution speed.[1] The sequential nature of the EVM execution client prevents modern multi-core processors from being fully utilized, capping execution throughput and stretching latency tails under network congestion.[1] Monad has emerged as an independent Layer 1 blockchain seeking to break this transaction throughput ceiling by delivering up to 10,000 transactions per second (TPS) with a 400 ms block frequency and an 800 ms finality time, while maintaining 100% bytecode compatibility with Ethereum.[4]

Founded by former Jump Trading engineers Keone Hon and James Hunsaker, Monad applies high-frequency trading (HFT) optimization principles to blockchain architecture.[6] This high performance is achieved by rearchitecting the execution, storage, and consensus pipelines through five major innovations: a custom MonadBFT consensus engine, a RaptorCast block propagation protocol, asynchronous execution, optimistic parallel transaction processing, and a native Patricia trie state database known as MonadDB.[3] However, pushing physical hardware and distributed network limits to these extremes introduces deep systemic trade-offs, physical performance cliffs, and unique engineering vulnerabilities. This analysis provides an exhaustive technical critique of Monad's architecture, evaluates its performance under high load, and identifies key pathways for technological optimization.

Consensus and Block Propagation Optimization: MonadBFT and RaptorCast

At the foundation of Monad's high-throughput architecture is MonadBFT, a Byzantine Fault Tolerant (BFT) consensus protocol derived from the HotStuff family.[4] Traditional BFT consensus algorithms, such as Practical Byzantine Fault Tolerance (PBFT) and Tendermint, exhibit quadratic communication complexity ($O(n^2)$) due to all-to-all communication topologies.[9] This design severely restricts the size of the active validator set, as messaging overhead becomes unsustainable when scaling beyond a few dozen nodes.[4] MonadBFT solves this scalability constraint by achieving linear message and authenticator complexity ($O(n)$) on the

happy path, enabling the protocol to support a decentralized network of 150 to 200 active validators on modest hardware.[4]

The protocol achieves this linear complexity by utilizing an optimistic, pipelined "one-to-many-to-one" communication structure.[6] In each consensus round (or view), a preselected leader proposes a block and collects cryptographic votes (attestations) from the validator set.[6] Rather than waiting for multiple round-trips to finalize a block before proposing the next one, MonadBFT pipelines the commit stages.[4] A block enters the consensus pipeline in each round; while block N is being finalized over successive rounds, the leader is already proposing and collecting votes for block $N + 1$.[4] This overlapping progression allows the protocol to operate at network latency, achieving speculative finality within a single round and complete cryptographic finality within two rounds (~800 ms).[4]

A significant architectural contribution of MonadBFT is its built-in resistance to "tail-forking".[8] In standard pipelined BFT protocols, if a leader goes offline or fails to collect a supermajority of votes, the previous unfinalized proposal is often abandoned.[8] Malicious leaders can exploit this vulnerability as a form of Maximal Extractable Value (MEV) attack, deliberately discarding their predecessor's block.[8] This deprives the previous proposer of rewards and allows the malicious leader to reorder, censor, or insert transactions across blocks to capture arbitrage opportunities.[8]

MonadBFT mitigates tail-forking by enforcing a strict block reproposal mechanism during view changes (the unhappy path).[8] When a consensus round fails due to network delays or an offline leader, validators include their "local tip" (the latest unfinalized proposal they voted for) in their timeout messages.[8] The next leader is mathematically obligated to aggregate these local tips, identify the "high tip" (the block from the highest view), and repropose it.[8] If the leader cannot recover the high-tip block, validators jointly issue a cryptographic No-Endorsement Certificate (NEC), which guarantees the block could never have achieved a Quorum Certificate (QC), allowing the leader to propose a new block safely.[13]

This high-frequency block generation creates a massive data propagation hurdle.[4] At 10,000 TPS and an average transaction size of 200 bytes, a validator must transmit 2 MB of block data per second.[4] Directly uploading a 2 MB block to 200 validators would require a leader to have 400 MB/s (3.2 Gbps) of upload bandwidth, which exceeds the capacity of standard validator hardware.[4] Monad resolves this bandwidth bottleneck through RaptorCast, a specialized block propagation protocol.[4]

RaptorCast rejects TCP in favor of UDP to minimize connection overhead and lower transmission latency.[14] It employs erasure coding based on Raptor codes, where a block is split into a series of smaller chunks.[4] The total size of all chunks is greater than the original block, but the original block can be fully reconstructed from any subset of chunks whose aggregate size matches the block's original volume (e.g., reclaiming a 1000 KB block from any 50 out of 150

generated chunks).[10] RaptorCast distributes these chunks using a structured, two-level broadcast tree.[4] The leader distributes unique chunks to "level-1" validators (assigned proportionally to their stake weight), who then rebroadcast their assigned chunk to all "level-2" validators.[6] This design limits worst-case propagation time to twice the worst-case one-way network latency while keeping the leader's upload bandwidth independent of the total validator count.[14]

Parameter	Traditional PBFT / Tendermint	MonadBFT / RaptorCast
Communication Complexity	Quadratic ($O(n^2)$) [9]	Linear ($O(n)$) on happy path [4]
Active Validator Capacity	Low ($\leq$ nodes for acceptable latency) [9]	High (150–200 nodes supported) [4]
Commit Latency	Sequential, multi-round block intervals [15]	Pipelined, single-round speculative, two-round final [4]
Block Time / Finality	12 seconds (Ethereum) [6]	400 ms block / 800 ms finality [4]
MEV Tail-Forking Resistance	Vulnerable (Unfinalized blocks easily discarded) [8]	High (Enforced reproposal of high tip via TCs/NECs) [8]
Transmission Protocol	TCP gossip networks (High latency hops) [14]	UDP Raptor code erasure chunking [10]
Required Leader Upload Bandwidth	Scaled linearly with validator count ($O(n)$)	Scaled independently of validator count via 2-level tree [10]

The Deferred Execution Pipeline and the RPC Consistency Paradox

A core structural limitation of legacy EVM blockchains is the coupling of consensus and execution.[12] In Ethereum, the block proposer must execute all transactions in a proposed block before sharing the proposal, and validating nodes must execute them before responding with

their vote.[15] This design squeezes the available execution window, forcing the system to allocate most of the block time to network communication and safety margins rather than computing transactions.[6] Monad decouples consensus from execution through an asynchronous execution model.[6]

In Monad, validators reach consensus on the linear ordering of transactions in a block *before* executing them.[5] Once transaction ordering is finalized via MonadBFT, execution occurs asynchronously on a separate pipeline, allowing execution to utilize the full block time without blocking consensus.[4] While execution workers process block N, consensus can proceed with voting on and proposing block $N+1$.[3] This decoupling allows Monad to increase its block gas limit to 150 million gas in testnet (equivalent to a gas rate of 375 million gas/second) and support highly complex smart contracts.[4]

However, this asynchronous execution model introduces a severe systems engineering challenge: the **loss of immediate read-your-own-writes (RYW) consistency** at the JSON-RPC layer.[17] In a standard EVM node, the state database is updated in lockstep with consensus, ensuring that client reads and writes are perfectly synchronized.[15] Because Monad defers execution, the state database on standard nodes lags behind consensus.[3] This replication lag breaks immediate RYW consistency, which is a core expectation for client-side applications, trading bots, and keepers.[17]

When a user submits a transaction via eth_sendRawTransaction, the RPC node cannot immediately validate nonces or balances because execution is lagging.[17] Consequently, invalid transactions (e.g., those with insufficient gas or incorrect nonces) are accepted into the mempool and only rejected much later during execution, complicating client-side error handling.[17] Furthermore, standard Geth RPC queries like eth_getTransactionByHash return null for pending transactions, as the transaction lifecycle behaves differently under the deferred execution model.[17]

If a trading bot submits a swap transaction and immediately queries the state database to read its new balance or update its pricing models, the query may route to a read replica or an execution thread that is still processing the previous block's state delta.[17] This lag breaks read-after-write verification, leading to stale reads and false-negative balances.[18]

This execution lag also impacts historical data access and event indexing.[17] Because Monad's high throughput generates up to 6TB of historical data per day, standard full nodes aggressively prune state, retaining only a rolling window of approximately 40,000 blocks (~4.5 hours of history).[17]

Consequently, deep historical eth_call and state queries are unavailable on standard nodes.[17] Furthermore, eth_getLogs requests are severely throttled to tight block ranges (typically 100 to 1,000 blocks per request) to prevent concurrent queries from overwhelming the node's memory.[17] This forces developers to rely on specialized, sharded archive architectures (such as

Goldsky Edge) to index and serve historical data, introducing external dependencies into the developer workflow.[20]

RPC API Feature	Ethereum (Synchronous EVM)	Monad (Asynchronous Deferred EVM)
Execution State Sync	Strictly synchronous (State updates with consensus) [15]	Eventually consistent (State lags behind consensus) [3]
Transaction Submission	Real-time state checks (Nonces and balances validated instantly)	Speculative state checks (Validation occurs post-consensus) [10]
eth_getTransactionByHash	Returns transaction details and status immediately	May return null for pending transactions during execution lag [17]
Local Node State Depth	Full state history retained by default	Aggressively pruned (Only ~40,000 blocks retained) [17]
eth_getLogs Range Limit	High (Often up to 10,000+ blocks per request)	Heavily restricted (100–1,000 blocks per request) [17]
Read-Your-Own-Writes	Guaranteed on the same node	Broken without session pinning or sticky routing [18]
Tooling Dependency	Low (Standard Geth RPC is sufficient)	High (Requires specialized indexers like Goldsky Edge) [20]

Dynamic Parallel Execution and Runtime Re-Execution Overheads

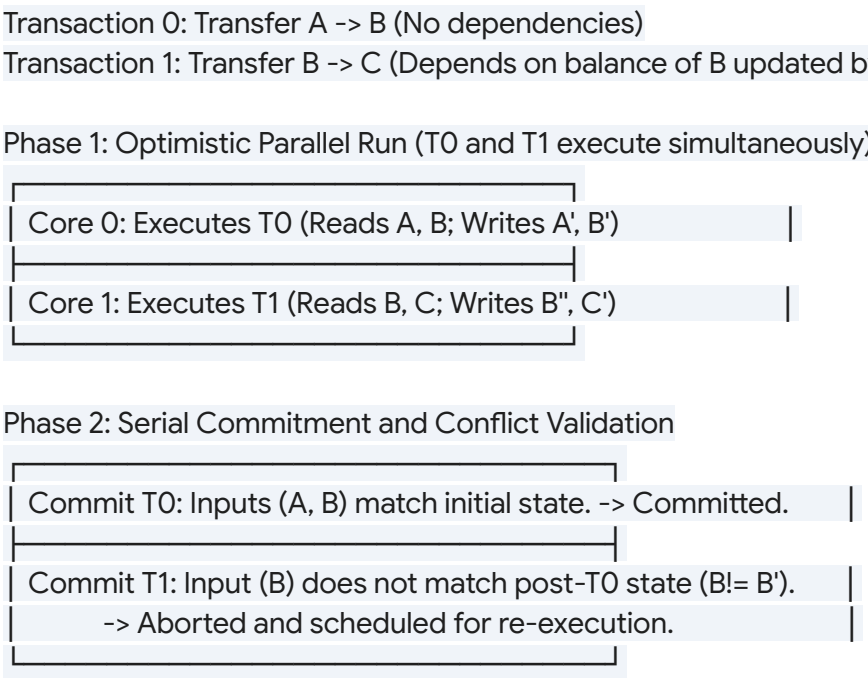
To maximize CPU utilization, Monad implements an Optimistic Concurrency Control (OCC) model for its parallel execution engine.[21] Transactions in a block are assumed to be independent and are processed in parallel across multiple CPU cores.[2] This dynamic approach stands in sharp contrast to static parallel execution models, such as Solana's Sealevel and Sui's object model, which require transactions to explicitly declare all read and write dependencies

upfront.[2]

While Solana and Sui avoid runtime conflicts by scheduling independent transactions in advance, they force developers to abandon standard EVM designs and rewrite contracts to handle explicit state declarations.[1] Monad preserves the linear transaction ordering of Ethereum.[4] It dynamically parallelizes execution without requiring any code changes from developers.[2]

During parallel execution (Phase 1), Monad's runtime schedules transactions across separate threads, recording all read operations (inputs) and write operations (outputs) in a temporary memory structure called a PendingResult.[25] During the serial commit phase (Phase 2), transaction outputs are sequentially merged into the global state.[21] Before applying a transaction's updates, the system checks whether its inputs match the current state.[25] If no prior transaction in the sequence has modified the state keys read by this transaction, the outputs are committed.[25]

If a conflict is detected (i.e., a prior transaction has modified a state key that a subsequent transaction read), the subsequent transaction's execution is aborted and re-executed with the updated state.[21] This dynamic dependency management ensures that the final state matches the result of sequential execution.[5]



While Monad minimizes re-execution overhead by caching state in memory and avoiding

re-running non-state-dependent tasks (such as signature recovery) [21], dynamic tracking still introduces significant runtime computation.[22]

In highly active DeFi environments with frequent shared state modifications (such as liquidity pools, shared AMM states, or liquidations), state hotspots are common.[26] When multiple transactions target the same state keys, Monad's parallel execution model can suffer from high abort rates.[26]

Under heavy state contention, the sequential validation phase is forced to discard the optimistic parallel runs and execute transactions sequentially.[26] This sequential bottleneck can drag down performance, potentially canceling out the benefits of parallel execution.[26]

To analyze how Monad compares to other high-performance blockchains, the table below outlines the trade-offs of different execution models:

Dimension	Solana (Sealevel)	Sui (Move VM)	SupraBTM (iBTM)	Monad (Parallel EVM)
Dependency Management	Static (Pre-declared in transaction signatures) [2]	Static (Object ownership metadata) [23]	Static deployment analysis & pre-run DAG [26]	Dynamic (Dynamic tracking at runtime) [24]
State Paradigm	Account-based (Global state) [23]	Object-centric (Owned/Shared objects) [23]	Account-based (EVM-native) [26]	Account-based (EVM-equivalent) [5]
Conflict Resolution	Prevented upfront by block scheduler [24]	Prevented upfront (No shared-state locks) [23]	Eliminated upfront via DAG scheduling [26]	Runtime abort, rollback, and sequential retry [22]
Re-Execution Risk	None [24]	None [23]	Minimal [26]	High under high state contention [26]
High Contention Performance	Stable (Bounded by scheduler queues)	High (Optimized for independent transactions) [23]	High (Dynamic fallback to sequential) [26]	Degrades to sequential speed with high compute overhead [26]

Developer UX	High complexity (Must declare all accounts upfront) [2]	High complexity (Must adapt to object ownership rules) [28]	Low complexity (Standard EVM compatibility)	Low complexity (100% bytecode and tooling equivalence) [3]

MonadDB: Custom Radix Tree and Filesystem Bypass

Execution speed is wasted if storage cannot keep up.[1] Most traditional Ethereum clients use generic key-value databases, such as LevelDB or RocksDB (implemented as LSM-Trees) or LMDB (implemented as B-Trees), to store state.[30] However, Ethereum structures its state as a Merkle Patricia Trie (MPT) to generate cryptographic proofs.[30]

This creates a suboptimal architecture where the MPT data structure is embedded inside a B-Tree or LSM-Tree database.[30] To read a single state value, the database must perform nested disk reads across multiple levels of the LSM-Tree or B-Tree, creating a severe I/O bottleneck.[4]

Traditional EVM Storage (Double Layer Indirection):

```
┌──────────────────────────────────────┐
│ State Query -> MPT Path Traversal    │
├──────────────────────────────────────┤
│ -> Embedded inside LevelDB/RocksDB LSM-Tree │
├──────────────────────────────────────┤
│ -> Traverses multiple LSM SST levels on disk │
└──────────────────────────────────────┘
```

MonadDB Storage (Native Single Layer Indirection):

```
┌──────────────────────────────────────────────┐
│ State Query -> Native Radix Tree (In-Memory/On-Disk) │
├──────────────────────────────────────────────┤
│ -> Direct asynchronous I/O (io_uring) on block device │
└──────────────────────────────────────────────┘
```

Monad resolves this storage bottleneck through MonadDB, a custom state database designed specifically for storing MPT nodes on disk.[30] MonadDB implements a Patricia Trie natively, both on-disk and in-memory, eliminating the double-layer indirection of traditional databases.[4]

Furthermore, to enable concurrent parallel execution, reads must be non-blocking.[30] Traditional databases lack proper asynchronous I/O (async I/O) support, which forces clients to spawn large numbers of kernel threads to handle pending I/O requests.[30] MonadDB fully utilizes the

latest Linux kernel support for async I/O via the io_uring interface, allowing the database to execute concurrent reads without blocking execution threads or incurring context-switching overhead.[30]

To minimize overhead further, MonadDB provides operators the option to bypass the filesystem.[30] Modern filesystems introduce significant performance costs due to block allocation, fragmentation, read/write amplification, and metadata management.[30] Bypassing the filesystem and running MonadDB directly on raw block devices allows the database to fully utilize SSD hardware performance.[30]

MonadDB implements its own indexing system based on the Patricia trie structure, coordinating concurrent access between a single writer (execution) and multiple readers (consensus and RPC).[30] This native architecture reduces the number of page reads from SSDs to perform a lookup, ensuring that disk reads (costing 40-100 microseconds) can execute in parallel without stalling execution threads.[4]

Storage Parameter	**LevelDB / RocksDB (Traditional Clients)**	**MonadDB (Custom Radix Store)**
Data Structure Model	Embedded LSM-Tree (Nested indirection) [30]	Native Radix Tree (No nested indirection) [4]
State Format Compatibility	Flat key-value store, MPT layered on top [30]	Optimized natively for Merkle Patricia Trie nodes [30]
Async I/O Support	Limited (Spawns synchronous kernel thread pools) [30]	Native async I/O (Leverages Linux io_uring) [30]
OS Filesystem Dependency	Mandatory (Interacts through system file calls) [30]	Optional (Bypasses filesystem; runs on raw block devices) [30]
SSD Performance Access	Squeezed by OS file locks, fragmentation, amplification [30]	Direct raw block access (Maximizes SSD performance) [30]
Concurrent Access Model	High lock contention under multi-threaded loads	Single writer (Execution), multiple readers (Consensus, RPC) [30]

Disk Read Latency	High (Cumulative latency over multiple index layers)	40–100 microseconds per native parallel read [31]

The Monad Gas Model: DoS Prevention and Economic Alignment

To support asynchronous execution, Monad cannot use Ethereum's standard gas billing model.[32] In Ethereum, a transaction is billed based on the exact amount of gas consumed during execution.[32] In Monad's asynchronous model, block leaders must build blocks and reach consensus on transaction ordering *before* those transactions are executed.[32]

If Monad billed based on actual gas used, a malicious actor could submit transactions with very high gas limits that consume virtually no gas.[32] This would allow them to pack a block to its gas limit during the consensus phase without paying for the reserved space, creating a severe Denial of Service (DoS) vulnerability.[32]

To resolve this, Monad implements a unique billing rule: **users are charged for the entire gas limit specified in the transaction, regardless of actual consumption**.[32]

$$\text{gas\_paid} = \text{gas\_limit} \times \text{price\_per\_gas}$$

This model prevents DoS attacks by forcing users to pay for any block space they reserve.[32] However, this shift introduces significant friction for developers and users. If a transaction reverts during execution (e.g., due to slippage limits or a sold-out NFT), standard EVM wallets like MetaMask fallback to estimating an extremely high gas limit.[32] In Ethereum, the user only pays for the gas used up to the reversion point. In Monad, the user is charged for the entire, inflated gas limit, resulting in severe and unexpected financial penalties.[32]

Additionally, during protocol deployments, asynchronous execution delays can cause transactions to be rejected by the mempool if they use standard EVM gas estimation limits.[17] Developers are forced to apply massive gas estimate overrides (such as using a --gas-estimate-multiplier 200 override) to ensure successful execution, creating unpredictable deployment costs.[17]

To align execution costs with physical hardware constraints, Monad has reweighted the gas pricing of cold state access opcodes.[32] While Monad's execution engine makes computation up to 300x cheaper, the physical cost of reading data from disk (state reads) remains bounded by SSD hardware speeds.[32]

If Monad kept Ethereum's original gas weights, storage would become 300x cheaper in relative terms, making the network highly vulnerable to storage-based DoS-style spam attacks.[32] To

prevent this, Monad has adjusted cold state access costs upward to reflect the physical cost of disk reads.[32] This is summarized in the opcode repricing matrix below [32]:

Opcode / Precompile	Affected Instructions	Ethereum Gas Cost	Monad Gas Cost	Multiplier
Cold Account Access	BALANCE, EXTCODESIZE, EXTCODECOPY, EXTCODEHASH, CALL, CALLCODE, DELEGATECALL, STATICCALL, SELFDESTRUCT	2,600	**10,100**	3.88x
Cold Storage Access	SLOAD, SSTORE	2,100	**8,100**	3.85x
Warm Account Access	BALANCE, EXTCODESIZE, etc. (Warm state cache)	100	**100**	1.0x (No change)
Warm Storage Access	SLOAD, SSTORE (Warm state cache)	100	**100**	1.0x (No change)
ecRecover	0x01 precompile (Signature recovery)	3,000	**6,000**	2.0x

ecAdd	0x06 precompile (Elliptic curve addition)	150	**300**	2.0x
ecMul	0x07 precompile (Elliptic curve multiplication)	6,000	**30,000**	5.0x
ecPairing	0x08 precompile (Elliptic curve pairing)	45,000	**225,000**	5.0x
blake2f	0x09 precompile (Cryptographic hashing)	rounds × 1	**rounds × 2**	2.0x
point eval	0x0a precompile (Blob transaction verification)	50,000	**200,000**	4.0x

To manage base fee adjustments at 400 ms block intervals, Monad uses a variance-aware controller inspired by adaptive optimization algorithms like RMSprop and Adam.[32] This controller is defined by the following mathematical equations [32]:

$$\text{block\_gas}_k = \sum_{\text{tx} \in \text{block}_k} \text{gas\_limit}_{\text{tx}}$$

$$\text{base\_price\_per\_gas}_{k+1} = \max\left\{\text{min\_base\_price\_per\_gas}, \text{base\_price\_per\_gas}_k \cdot \exp\left(\eta_k \cdot \frac{\text{b}}{\text{blo}}\right.\right.$$

The dynamic learning rate (η_k), which dampens step size adjustments under volatile demand, is calculated using the historical trend and moment of gas usage [32]:

$$\eta_k = \frac{\text{max\_step\_size} \cdot \epsilon}{\epsilon + \sqrt{\text{moment}_k - \text{trend}_k^2}}$$

$$\text{trend}_{k+1} = \beta \cdot \text{trend}_k + (1 - \beta) \cdot \left(\text{target} - \text{block\_gas}_k\right)$$

$$\text{moment}_{k+1} = \beta \cdot \text{moment}_k + (1 - \beta) \cdot \left(\text{target} - \text{block\_gas}_k\right)^2$$

Here, the target utilization is set to 80% (160 million gas) of the 200 million gas block limit, the smoothing factor β is set to 0.96 (equivalent to a 17-block half-life), and the maximum step size is capped at $1/28$.[32] This mathematical model prevents the base fee from becoming too volatile ("twitchy") during short demand spikes, but it increases fee complexity and makes it harder for developers to predict transaction costs.[32]

State Expansion, Hardware Demands, and Node Centralization

With a testnet block gas limit of 150 million gas and 400 ms block times, Monad processes an unprecedented 375 million gas per second.[4] Under production workloads, this throughput will accelerate state expansion to extreme levels.[1] State expansion refers to the accumulation of contract code, accounts, and storage slots that must be held in a node's active memory or storage to validate transactions.[33]

Monad's high write throughput can generate up to 6TB of historical archive data per day.[20] This state expansion creates a massive operational footprint for node operators, leading to serious performance degradation and node centralization.[1]

To achieve sufficient state read/write speeds, Monad enforces demanding hardware requirements.[35] Operating a node requires a dedicated, bare-metal server; virtualization (e.g., AWS, GCP, Azure) is strictly discouraged due to the latency and context-switching overhead introduced by hypervisors.[35]

Furthermore, standard full nodes only retain a rolling window of approximately 40,000 blocks of state history (about 4.5 hours of operations) before aggressively pruning.[17] A historical archive node, which is essential for developer indexing, blockchain explorers, and historical analytics, requires over 16TB of NVMe storage and takes up to a week to bootstrap and sync.[17]

The physical limits of Monad's hardware profiles are detailed below:

Technical Parameter	Full Node Specification	Active Validator Node	Archive Node Specification
CPU Architecture	16-Core, $\geq$ Base (e.g., AMD Ryzen 9950x) [35]	16-Core, $\geq$ Base (e.g., AMD Ryzen 9950x) [35]	16-Core, $\geq$ Base (e.g., AMD Ryzen 9950x) [35]
System Memory	$\geq$ RAM [35]	$\geq$ RAM [35]	$\geq$ RAM [35]
Storage Configuration	2TB PCIe Gen4x4 NVMe SSD for TrieDB [35]	2TB TrieDB NVMe SSD + 500GB MonadBFT NVMe SSD [35]	$\geq$ Dedicated Enterprise NVMe SSD [17]
Bandwidth (Ingress/Egress)	$\geq$ Sustained [4]	$\geq$ Sustained [4]	$\geq$ Dedicated Burstable
Virtualization Compatibility	Unstable (Highly discouraged) [35]	Incompatible (Slashes performance) [35]	Incompatible (Extreme hypervisor I/O bottlenecks)
Consensus Stake Threshold	N/A (Non-voting listener) [10]	$\geq$ Total Stake (Top 200 restriction) [12]	N/A (Query service layer)
Bootstrap / Sync Duration	$\sim$	$\sim$	Up to 7 Days (Sustained I/O bottleneck) [17]

These demanding requirements introduce a clear decentralization ceiling.[1] The high cost of bare-metal hardware and dedicated bandwidth restricts node operation to professional data centers and well-capitalized institutions, which could lead to validator centralization.[1]

Furthermore, because Monad's active consensus validator set is limited to the top 200 nodes by total stake (with a minimum self-stake of 100,000 MON and a total stake requirement of 10 million MON) [12], block production is highly concentrated. If historical data access is restricted to a small number of centralized archive providers, the network's trustlessness could be

compromised.[20]

Nuanced Conclusions and Actionable Recommendations

The technological design of the Monad blockchain pushes physical hardware and distributed network limits to deliver massive transaction throughput.[3] However, this hyperscaling introduces deep systemic trade-offs, storage bottlenecks, and execution vulnerabilities.[1]

To optimize performance, reduce developer friction, and preserve decentralization, Category Labs should implement the following targeted engineering modifications:

1. Implement Proactive Conflict-Aware Scheduling

Monad's current optimistic execution model (2PE) suffers from high abort and re-execution rates under heavy state contention, dragging down performance to sequential speeds.[26]

To resolve this, Monad should integrate a proactive scheduling layer similar to SupraBTM.[26] Before executing transactions, a lightweight scheduler should analyze transaction signatures and mempool state to detect conflicts.[26]

If the anticipated conflict ratio exceeds a dynamically calibrated threshold, the system should adaptively schedule the conflicting subset of transactions or group them into a dependency DAG prior to execution.[26] This hybrid approach would prevent threads from executing transactions that are guaranteed to abort, saving valuable CPU cycles under heavy load.[22]

2. Deploy Speculative RPC State-Tracking Proxies

The loss of immediate read-your-own-writes (RYW) consistency due to deferred execution creates major integration challenges for client-side applications, trading bots, and keepers.[17]

To restore seamless integration, Monad’s RPC nodes should implement a local, speculative state-tracking proxy.[19] This proxy would maintain a short-lived memory projection of pending transactions, routing read requests (such as eth_call) through a speculative execution sandbox that applies the client's pending transactions onto the current committed state.[19]

Additionally, session-based routing should be enforced to pin a client's read requests to the same node that processed their writes, guaranteeing immediate local consistency without impacting global validator performance.[18]

3. Integrate Native State Expiry and Eviction Protocols

The high write throughput of Monad’s 375 million gas/second rate accelerates state expansion, creating severe storage and bandwidth barriers for node operators.[1]

To prevent runaway state bloat and reduce hardware requirements, Monad should build an

active state-expiry protocol directly into MonadDB.[30] Accounts and contract storage keys that have not been accessed within a configured epoch window (e.g., 30 days) should be evicted from the active on-disk Patricia Trie and moved to cold, decentralized storage layers.[34]

To access these accounts, users would submit a cryptographic proof of the historical state alongside their transaction.[34] This active pruning would cap the active TrieDB storage requirement to a predictable size, allowing validators to run on affordable commodity NVMe drives without losing performance.[4]

4. Provide Post-Execution Gas Refunds

Charging users for their entire gas limit (rather than actual consumption) prevents DoS attacks under deferred execution, but it penalizes honest users when transactions revert due to standard conditions.[32]

To reduce this friction, Monad should implement a secure post-execution gas refund mechanism.[32] During the serial commit phase (Phase 2), if a transaction is proven to be honest (e.g., it did not revert due to state-exhaustion attacks and actual consumption was far below the specified limit), the protocol should refund up to 50% of the unused gas limit directly to the user's balance.[25]

This would protect users from severe over-billing while maintaining robust DoS protection across the network.[32]

Works cited

1. What Is Monad? The High-Throughput EVM Layer-1 With Parallel Execution - Atomic Wallet, accessed May 17, 2026, https://atomicwallet.io/academy/articles/what-is-monad
2. Parallel Execution & Monad, accessed May 17, 2026, https://www.monad.xyz/announcements/parallel-execution-monad
3. Monad Review: The fast parallel EVM - Rango Exchange, accessed May 17, 2026, https://rango.exchange/learn/market-trends/monad-blockchain-review
4. How Monad Works, accessed May 17, 2026, https://blog.monad.xyz/blog/how-monad-works
5. Monad for Developers - Monad Documentation, accessed May 17, 2026, https://docs.monad.xyz/introduction/monad-for-developers
6. Monad: A Supercharged EVM Layer 1 - Blockworks Research, accessed May 17, 2026, https://app.blockworksresearch.com/unlocked/monad_a-supercharged-evm-layer-1
7. High-Performance EVM Layer-1 for Scalable DeFi - Infura, accessed May 17, 2026, https://www.infura.io/networks/monad
8. MonadBFT: Fast, Responsive, Fork-Resistant Streamlined Consensus - arXiv, accessed May 17, 2026, https://arxiv.org/html/2502.20692v2
9. MonadBFT Analysis (Part 1): Resolving Tail Fork Issues | Gate Learn, accessed May

17, 2026, https://www.gate.com/learn/articles/monad-bft-analysis-part-1-resolving-tail-fork-issues/8817
10. How Monad Works - HackMD, accessed May 17, 2026, https://hackmd.io/@keone/how-monad-works
11. MonadBFT: Fast, Responsive, Fork-Resistant Streamlined Consensus - ResearchGate, accessed May 17, 2026, https://www.researchgate.net/publication/389510445_MonadBFT_Fast_Responsive_Fork-Resistant_Streamlined_Consensus
12. Monad First Look: Hyperscaling the EVM - Figment.io, accessed May 17, 2026, https://www.figment.io/insights/monad-first-look-hyperscaling-the-evm/
13. MonadBFT: Fast, Responsive, Fork-Resistant Streamlined Consensus - arXiv, accessed May 17, 2026, https://arxiv.org/pdf/2502.20692
14. Monad's Road to Mainnet | Everstake, accessed May 17, 2026, https://everstake.one/resources/blog/monads-road-to-mainnet
15. Pioneering Parallelisation with Monad - Luganodes, accessed May 17, 2026, https://luganodes.com/blog/PioneeringParallelisationMonad
16. Monad Deferred Execution: Unlocking Scalability and Efficiency in Blockchain Transactions, accessed May 17, 2026, https://medium.com/@kintsu_xyz/monad-deferred-execution-unlocking-scalability-and-efficiency-in-blockchain-transactions-98afe27f38d8
17. AL Technical Analysis Aave V3.6 <> Monad - Risk - Aave, accessed May 17, 2026, https://governance.aave.com/t/al-technical-analysis-aave-v3-6-monad/24402
18. Read-Your-Writes Consistency - Arpit Bhayani, accessed May 17, 2026, https://arpitbhayani.me/blogs/read-your-write-consistency/
19. A Deep Dive on Read Your Own Writes Consistency - DZone, accessed May 17, 2026, https://dzone.com/articles/read-your-own-writes-consistency
20. How Monad handles 10K+ RPS when legacy infrastructure won't work - Goldsky, accessed May 17, 2026, https://goldsky.com/case-studies/monad
21. Parallel Execution - Monad Documentation, accessed May 17, 2026, https://docs.monad.xyz/monad-arch/execution/parallel-execution
22. Parallel Transaction Execution in Blockchains | Vlad Temian - Centrist Tech Optimist, accessed May 17, 2026, https://blog.vtemian.com/post/parallel-transaction-execution-in-blockchains/
23. Sui vs Solana: Comprehensive Blockchain Comparison & Analysis - Ledger, accessed May 17, 2026, https://www.ledger.com/academy/topics/blockchain/sui-vs-solana
24. [CryptoTimes] Blockchain & Technology Enabling Parallel Execution - DeSpread Research, accessed May 17, 2026, https://research.despread.io/crypto-times-parallel-execution/
25. Deep Dive into Monad's Architecture - Chorus One, accessed May 17, 2026, https://chorus.one/articles/deep-dive-into-monads-architecture
26. Supra vs. Monad: Towards the Best Parallel Execution of the ..., accessed May 17, 2026, https://supra.com/academy/supra-vs-monad/
27. Monad Scanner. Monad Execution & State Risk Scanner... | by Chain Digital |

Medium, accessed May 17, 2026, https://medium.com/@chaindigital/monad-scanner-f0e4e59ba9c2
28. Solana vs. Aptos vs. Sui: Which Layer-1 Is Best Positioned for Growth? - SorooshX, accessed May 17, 2026, https://www.sorooshx.com/blog/solana-vs-aptos-vs-sui-layer-1-growth
29. What Is Monad: Complete Parallelized EVM Layer 1 Guide (2026) - by DEXTools, accessed May 17, 2026, https://www.dextools.io/tutorials/what-is-monad-parallelized-evm-l1-guide-2026
30. MonadDb - Monad Documentation, accessed May 17, 2026, https://docs.monad.xyz/monad-arch/execution/monaddb
31. Deep Dive into MonadDB. The limitations of traditional... | by MonadWhisper - Medium, accessed May 17, 2026, https://medium.com/@monadwhisper/deep-dive-into-monaddb-f3aa1c6532f5
32. Understanding Monad's Gas Model: Smarter, Faster, and Safer | by ..., accessed May 17, 2026, https://medium.com/@husseinrasti/understanding-monads-gas-model-smarter-faster-and-safer-3b2f6319ff32
33. How to Raise the Gas Limit, Part 1: State Growth - Paradigm, accessed May 17, 2026, https://www.paradigm.xyz/2024/03/how-to-raise-the-gas-limit-1
34. What is State Bloat in Blockchains? - Nervos Network, accessed May 17, 2026, https://www.nervos.org/knowledge-base/state_bloat_blockchain_(explainCKBot)
35. Hardware Requirements - Monad Documentation, accessed May 17, 2026, https://docs.monad.xyz/node-ops/hardware-requirements